華中科技大學

操作系统原理课程实验报告

姓　　名：　　杨释钧

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　CS2006

学　　号：　　U202015471

指导教师：　　李晓露

分数	
教师签名	

2022 年 12 月 29 日

目　录

实验一 复杂缺页异常

1.1 实验目的

本实验给出的应用程序在运行过程中会发生用户栈的缺页异常和非法地址访问，我们需要修改 pke 内核代码来对这两种异常做出正确的处理，也就是需要在 lab2_3 已完成的缺页异常的基础上增加对非法地址访问的处理。通过本次实验，我们可以知道操作系统中对复杂缺页异常处理的实现方法。

1.2 实验内容

本实验给出的应用程序是对给定 n 计算从 0 到 n 的和，要求用递归计算，将每一步计算结果保存到数组 ans 中，首先当 n 比较大的时候，在函数递归执行时会触发用户栈的缺页，我们需要对其正常处理以确保应用程序正确执行；其次当 n 比较大的时候会访问数组越界地址，由于该处虚拟地址尚未有对应的物理地址映射，因此属于非法地址访问，对于这种缺页异常，我们应该提示用户并退出程序执行。

根据上面对于实验任务的分析，可以知道我们这次任务的重点是判断造成缺页异常的原因是不是发生了非法地址访问，如果不是，就认为是用户栈的缺页，沿用 lab2_3 中的处理方法为其分配一个物理页即可，否则就用任务书中要求的处理方法，也即提示用户发生了非法地址访问，并且直接退出程序即可。

现在我们考虑一下怎么区分这两种缺页异常。根据任务书，我们知道当前函数的参数 stval 是发生缺页异常的时候程序想要访问的逻辑地址，而当前的栈指针 sp 储存的是栈顶地址，比栈顶地址更低的空间是堆空间，不能被访问，因此只需要判断一下当前的 stval 和 sp 的大小关系即可，如果访问的空间比 sp 更低，说明发生了非法访问，应该打印错误信息并退出，否则分配新的物理页即可。

最终核心代码如下所示。

```
if(stval>=current->trapframe->regs.sp&&stval-current->trapframe->regs.sp<32){
    map_pages(current->pagetable,stval-stval%4096,4096,(uint64)alloc_page(),pro
t_to_type(PROT_READ|PROT_WRITE,1));
}else{
    panic("this address is not available!");
}
```

1.3 实验调试及心得

本实验在理解了非法访问和用户栈缺页异常的区别之后整体还是比较简单的，此外也需要理解内存空间的布局，比如栈空间的特点等。在具体的实现过程中没有出现太大问题。

实验二 进程等待和数据段复制

1.1 实验目的

本实验需要我们实现系统调用 wait，通过修改 PKE 内核和系统调用为用户程序提供 wait 函数的功能，同时要完善 do_fork 函数中关于数据段复制部分的代码来保证 fork 后父子进程的数据段相互独立。通过本次实验，我们可以学习到在操作系统中增加一种系统调用的方法，同时对 wait 和 fork 的知识有更深一步的理解。

1.2 实验内容

首先先来看本实验中实现 wait 的部分，该 wait 函数接收一个参数 pid，需要实现如下功能：

1. 当 pid 为-1 时，父进程等待任意一个子进程退出并返回该子进程号；
2. 当 pid 大于 0 时，父进程等待进程号为 pid 的子进程退出并返回 pid；
3. 当 pid 不合法或 pid 大于 0 且 pid 对应的进程不是当前进程的子进程时，需要返回-1；

我们从用户调用的 wait 函数开始一点点介绍这一部分的实现。这里主要参考了其它系统调用如 fork、yield 的写法，加入一个 wait 函数，wait 函数的主体是一个死循环，在内部进行系统调用，我们这里写的内部的系统调用如果在子进程正在运行的话就返回-2，这个时候我们需要调用 yield 将当前进程的 CPU 使用让出去；否则就正常返回。最终代码如下。

```
int wait(int pid){
    while(1){
        int rt=do_user_call(SYS_user_wait,pid,0,0,0,0,0,0);
        if(rt>-2)
            return rt;
        yield();
    }
}
```

接下来仿照 fork，增加 SYS_user_wait 等字段。然后就进入了对 PKE 内核代码的修改，首先要在 do_syscall 中仿照其它系统调用增加一个 case，实现一个 sys_user_wait 函数，实际上调用了 do_wait 函数，该函数就是我们最终要实现的

wait 的核心。另外这些参考着写的代码感觉意义不是很大，因此在这里不再引用。

然后介绍一下 do_wait 函数的实现，其核心思路也比较简单，只需要把任务书中提到的功能都实现出来即可：

1. 当 pid 为-1 时，需要返回一个退出的子进程，这个时候需要遍历进程表 procs，如果是当前进程的子进程并且其状态是僵尸状态 ZOMBIE 的话就将其释放掉，这里简单地将其状态改成 FREE 即可；如果当前进程有子进程并且都不需要释放的话，这个时候我们需要保存当前进程的信息，因此仿照了就绪队列 ready_queue 写了一个等待队列 wait_queue，如果没有僵尸进程的话就把 current 进程加入到等待队列里；这种情况还有一个需要注意的点，就是需要判断当前的进程有没有子进程。
2. 当 pid 大于 0 并且小于 NPROC 时，需要判断 pid 对应的进程的父进程是不是 current，如果不是直接返回-1，如果是的话需要对其状态进行判断，具体而言和第 1 种情况一样，此处不再赘述。
3. 当 pid 不满足上面两种情况时，认为 pid 不合法，最后返回-1。

do_wait 函数的代码如下所示。

```
int do_wait(int pid){
  if(pid==-1){
    int i;
    int flag=0;
    for(i=0;i<NPROC;i++){
      if(procs[i].parent==current&&procs[i].status==ZOMBIE){
        procs[i].status=FREE;
        return i;
      }else if(procs[i].parent==current){
        flag=1;
      }
    }
    if(flag==1){
      insert_into_wait_queue(current);
      schedule();
      return -2;
    }else{
      return -1;
    }
   }else if(pid>=0&&pid<NPROC){
```

```
    if(procs[pid].parent!=current){
      return -1;
    }
    if(procs[pid].status==ZOMBIE){
    procs[pid].status=FREE;
    return pid;
    }else{
      insert_into_wait_queue(current);
      schedule();
      return -2;
    }
  }
  return -1;
}
```

接下来说一下 insert_into_wait_queue 的实现，实际上和 insert_to_ready_queue 函数的实现基本一模一样，只需要将后者的 READY 状态改成 BLOCKED 状态，然后自定义一个插入的队列即可，此处不再赘述。

在 do_wait 实现完毕之后，可以发现我们还没有实现在子进程退出之后通知父进程的机制。其实这个实现也是比较简单的，因为我们之前是在父进程发现子进程还在运行中的时候就把父进程加入到 wait_queue 中，那这里其实只需要把父进程再加入到 ready_queue 中即可，具体的实现也比较简单，我这里实现了一个 change_queue 函数，用来比对 wait_queue 中和当前进程的父进程一致的进程，然后将它加入到 ready_queue 中，在系统调用 exit 中执行该函数即可，代码如下。

```
void change_queue(process* proc){
  process* p=wait_queue_head;
  while(p){
    if(p==proc->parent){
      p->status=READY;
      insert_to_ready_queue(p);
      p=p->queue_next;
      return;
    }
    p=p->queue_next;
  }
}
```

自此，系统调用 wait 的实现已经完成了，最后我们需要完成系统调用 fork 中关于数据段复制部分的代码，由于 fork 要求子进程有自己单独的数据段，因此我们需要分配新的空间，也就是需要调用 alloc_page，然后用 memcpy 将数据段内容拷贝过去，基本的拷贝和映射可以参考之前的实验，总之难度不大，代码如下。

```
case DATA_SEGMENT:
        for(int j=0;j<parent->mapped_info[i].npages;j++){
          char* newDataSeg=alloc_page();
          memcpy(newDataSeg,(void*)lookup_pa(parent->pagetable,parent->mapped_info[i].va+j*4096),4096);
          map_pages(child->pagetable,parent->mapped_info[i].va+j*4096,4096,(uint64)newDataSeg,prot_to_type(PROT_WRITE|PROT_READ,1));
        }
        break;
```

自此，该实验所需要实现的功能就成功完成了。

1.3 实验调试及心得

本实验的难度相对较大一点，但是只要搞清楚 PKE 中其它系统调用的实现，并且跟着调用流程一步步添加系统调用 wait 的相关代码就可以，比较核心的就是 wait 本身的逻辑，也就是子进程正常运行时父进程阻塞以及子进程运行结束时父进程重新就绪这两块逻辑的实现，只要理清楚这一点的逻辑其实对于 wait 的实现也就没有什么很难的点了。对于 do_fork 的补充的话，其实本身难度也不是很大，想清楚数据段复制时要求的子进程和父进程的数据段隔离这一个要求就行，由于已经有了前面几个实验的铺垫，所以这一部分在实现的时候也不会很困难。